Shared Memory – OpenMP

Computer Architecture and Operating Systems Department
Universidad Autónoma de Barcelona
tomas.margalef@uab.es

Shared Memory

- OpenMP is currently the programming standard for the shared memory model on multicore systems.

Shared Memory

- Thread based model.
- Threads read and write shared variables.
- Synchronization mechanism are offered.
- It is possible to change the attributes of threads and data for minimizing synchronization.

OpenMP

- OpenMP is not an automatic parallelization tool:
  - Programmers must specify parallelism explicitly.
- OpenMP is not only for exploiting loop parallelism:
  - It also offers functionalities for other forms of parallelism.

OpenMP

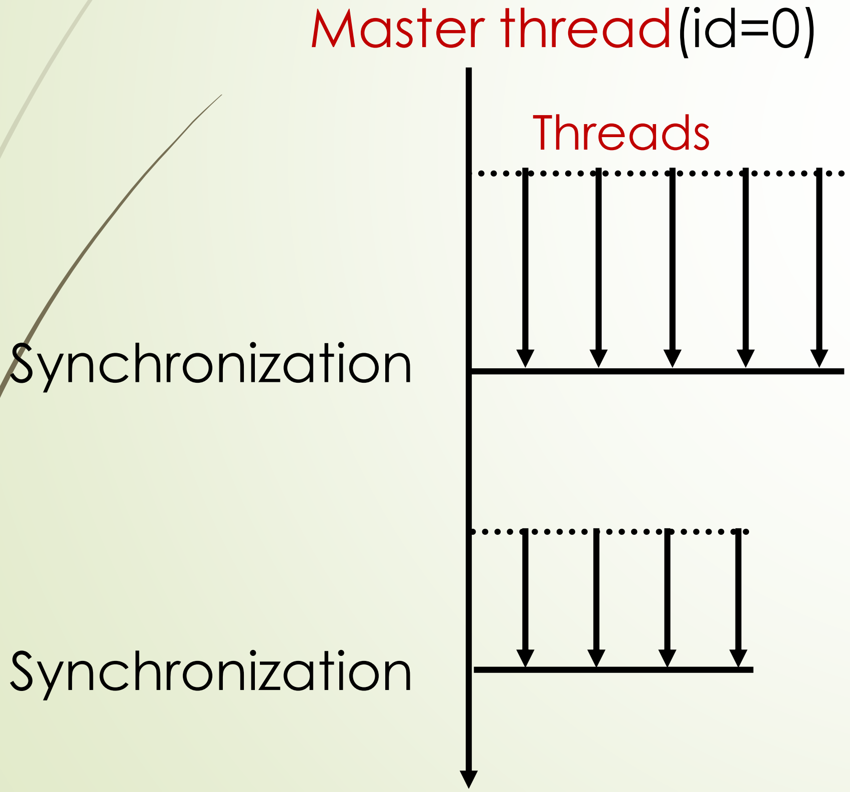
- OpenMP is not a programming language:
  - It is structured as extensions using directives to base languages like Fortran or C.
- OpenMP is not only a research project:
  - Many commercial compilers support OpenMP.

OpenMP

- OpenMP is an API (Application Program Interface) for parallel programming in shared memory systems.
- It's main objective is to easily parallelize existing applications.
- It's based in directives introduced in the program as special comments (pragmas).

OpenMP Execution Model

- Initially based on a FORK-JOIN model



!$OMP PARALLEL

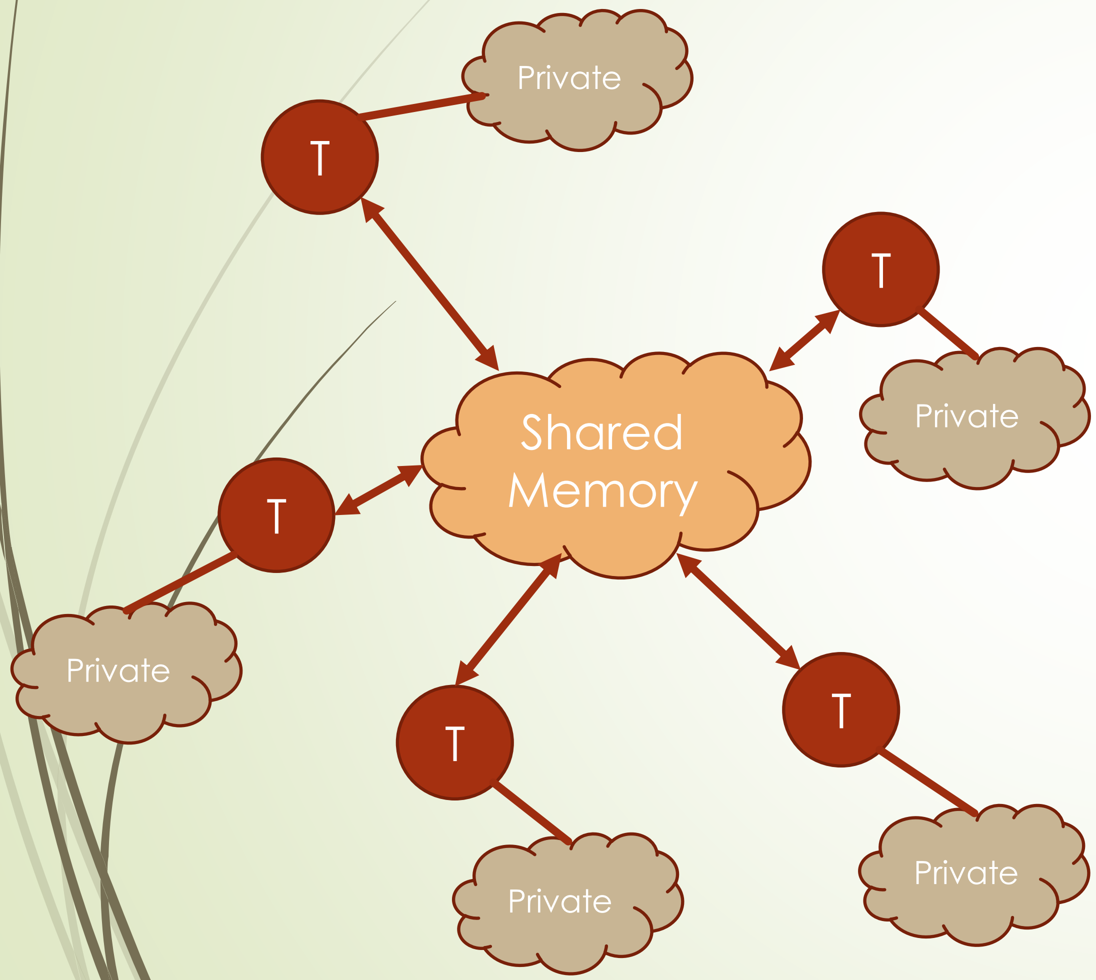
!$OMP END PARALLEL

!$OMP PARALLEL

!$OMP END PARALLEL

```
#pragma omp parallel
{

}

#pragma omp parallel
{

}
```

Shared Memory Model



- Data can be Shared or private.
- All threads have access to the same globally Shared memory.
- Shared data is accessible by all threads.
- Private data can be accessed only by the threads that owns it.
- Data transfer is transparent to the programmer.
- Synchronization takes place, but it is mostly implicit.

Shared Memory Model

- In a Shared memory parallel program variables have a "*label*" attached to them:

  - Labelled "*Private*" ➡ Visible to one thread only.
    - Change made in local data, is not seen by others.

  - Labelled "*Shared*" ➡ Visible to all threads.
    - Change made in global data, is seen by all others.

Components of OpenMP

- Directives
  - Parallel regions
  - Work sharing
  - Synchronization
  - Data scope attributes
    - private
    - firstprivate
    - lastprivate
    - shared

- Environement variables
  - Number of threads
  - Scheduling type
  - Dynamic thread adjustment
  - Nested parallelism

- Runtime Environement
  - Number of threads
  - Dynamic thread adjustment
  - Nested parallelism
  - Timers
  - API for locking

User Interface

- Compiler directives
  - There are control structures and data attributes structures.
  - Compilers ignore these directives (they are just comments) unless the proper options are used when compiling (“-mp” or “-fopenmp”).

User Interface

- Environment variables
  - Set of variables for controlling some parameters

```
setenv OMP_NUM_THREADS 8
```

- Runtime Library
  - Set of functions for controlling some parameters, such as the number of threads to be

```
omp_set_num_threads (128)
```

The omp parallel directive

- A parallel region is a block of code executed by multiple threads simultaneously.

```
#pragma omp parallel [clause[[,] clause] ...]
{
        "This will be executed in parallel by each thread"

}   (implied barrier)
```

C

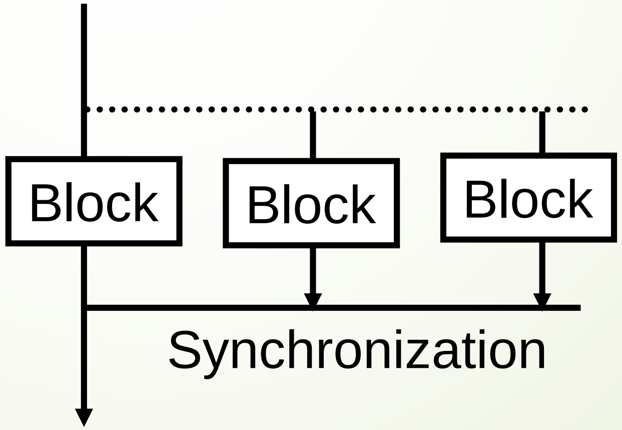
```
!$omp parallel [clause[[,] clause] ...]

        "This will be executed in parallel by each thread"

!$omp end parallel   (implied barrier)
```

Fortran

The omp parallel directive

- Define a parallel region.
- It does the “fork” and “join”.
- The number of threads is constant in the parallel region.

```
#pragma omp parallel{
        Block
}
```




```
!$OMP PARALLEL
        Block
!$OMP END PARALLEL
```

The omp parallel clauses

- omp parallel supports the following clauses:
  - if (scalar expression)
  - private (list)
  - shared (list)
  - default (none/shared)
  - reduction (operator: list)
  - firstprivate (list)
  - num_threads (scalar_int_expr)

The omp for/do directive

- The iterations of the loop are distributed over the threads.

```
#pragma omp for [clause[[,] clause] ...]
        <original for-loop>
```

C

```
!$omp parallel [clause[[,] clause] ...]
        <original do-loop>
```

Fortran

- Clauses supported:
  - private   firstprivate
  - reduction
  - schedule
  - nowait

The omp for/do directive

- #pragma omp for / !$omp do - !$omp end do
  - It is for classical parallel loops.
  - It must be in a parallel region.
  - Loop iterations are distributed among available threads.
  - Loop index is by default private to each thread.

The omp for directive - example

```
{
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;

    for (i=0; i<n-1; i++)
        d[i] = 1.0/c[i];
}
```

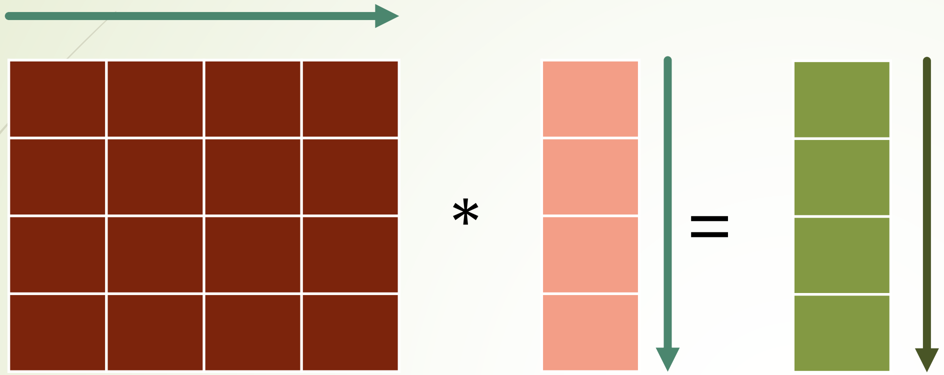
The omp for directive - example

```
#pragma omp parallel default (none) shared (n,a,b,c,d) private (i)
{


    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;


    for (i=0; i<n-1; i++)
        d[i] = 1.0/c[i];

}
```

The omp for directive - example

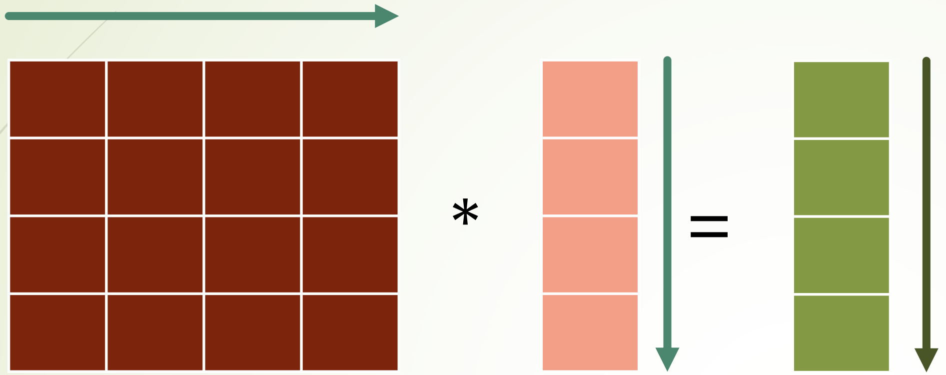
```
#pragma omp parallel default (none) shared (n,a,b,c,d) private (i)
{

    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        d[i] = 1.0/c[i];

}
```

Example: Matrix-Vector multiplication




```
for (i=0; i<m; i++)
{
    sum = 0.0;
    for (j=0; j<n; j++)
        sum += b[i][j]*c[j]
    a[i] = sum;
}
```

Example: Matrix-Vector multiplication



```
#pragma omp parallel for default (none) shared (m,n,a,b,c) private (i,j,sum)
for (i=0; i<m; i++)
{
    sum = 0.0;
    for (j=0; j<n; j++)
        sum += b[i][j]*c[j]
    a[i] = sum;
}
```

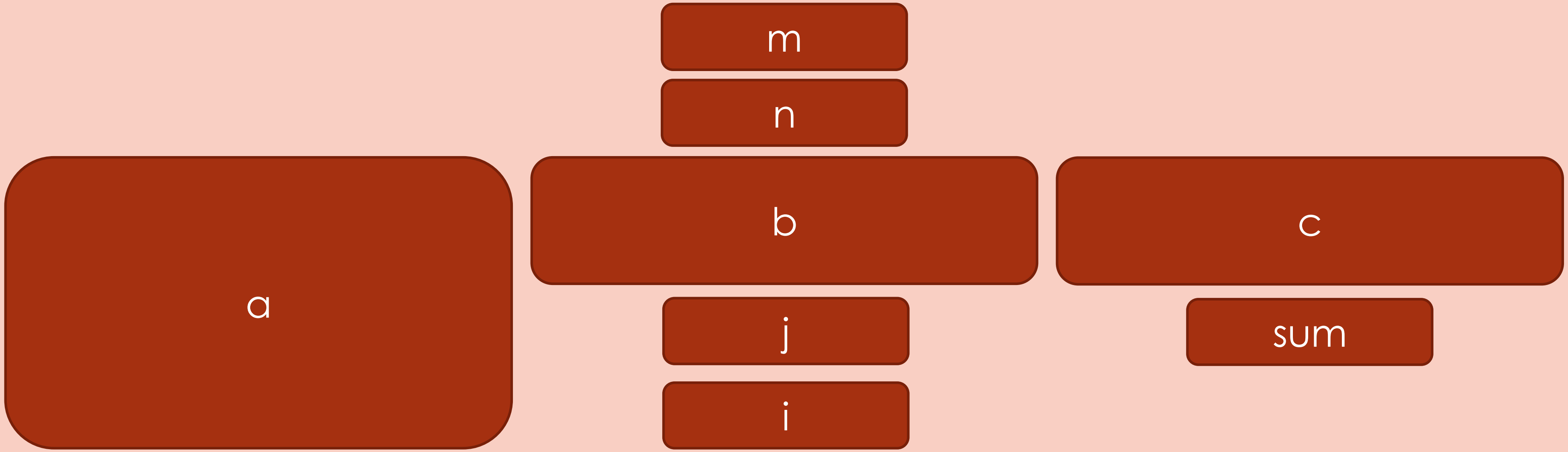
Example: Matrix-Vector multiplication

```
#pragma omp parallel for default (none) shared (m,n,a,b,c) private (i,j,sum)
for (i=0; i<m; i++)
{
 sum = 0.0;
 for (j=0; j<n; j++)
        sum += b[i][j]*c[j]
 a[i] = sum;
}
```

TID = 0	TID = 1	TID = 2	TID = 3
for (i = 0, 1, 2, 3, 4)	for (i = 5, 6, 7, 8, 9)	for (i =10,11,12,13,14)	for (i=15,16,17,18,19)
i = 0	i = 5	i = 10	i = 15
sum = ∑ b[0][j]*c[j]	sum = ∑ b[5][j]*c[j]	sum =∑b[10][j]*c[j]	sum =∑b[15][j]*c[j]
a[0] = sum	a[5] = sum	a[10] = sum	a[15] = sum
i = 1	i = 6	i = 11	i = 6
sum = ∑ b[1][j]*c[j]	sum = ∑ b[6][j]*c[j]	sum =∑b[11][j]*c[j]	sum =∑b[16][j]*c[j]
a[1] = sum	a[6] = sum	a[11] = sum	a[16] = sum
...	...	...	...

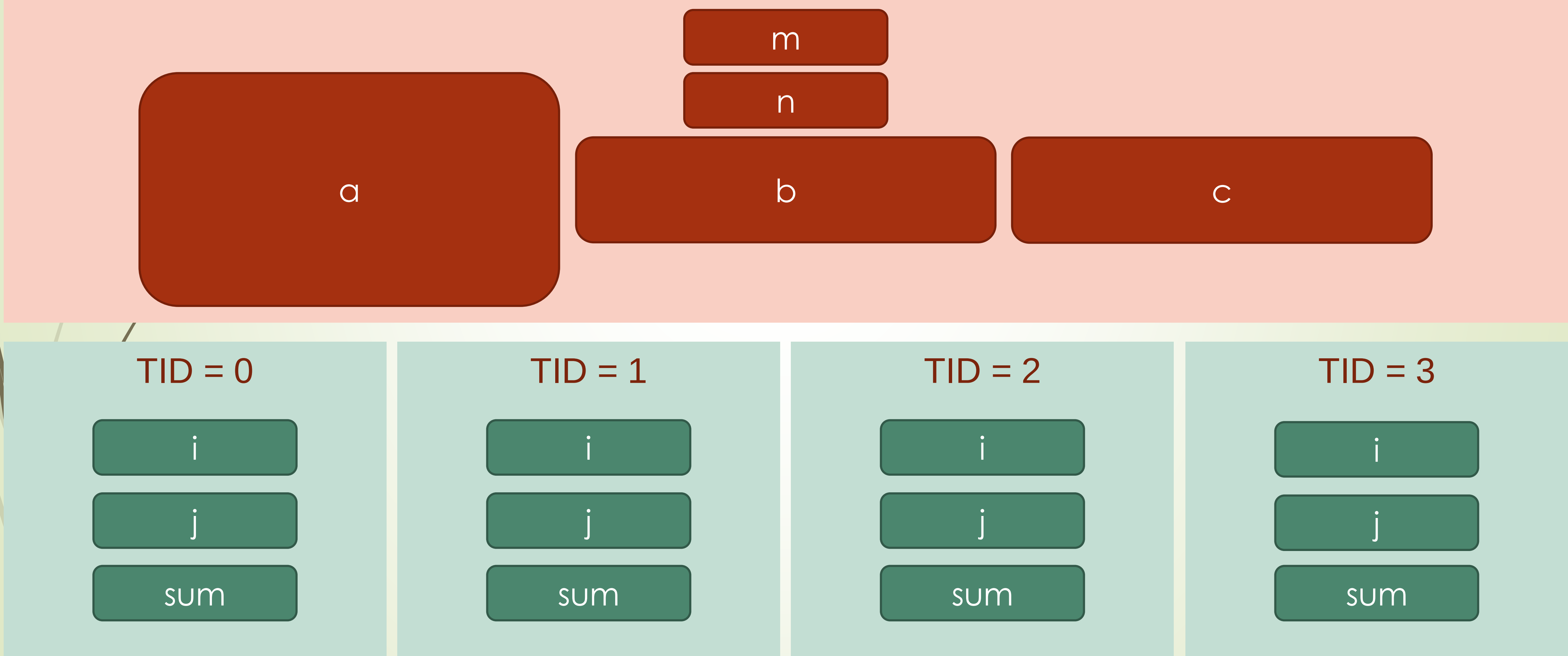
Example: Matrix-Vector multiplication

```
int m, n, i, j, sum
int a[100][100], b[100], c[100]
```



Example: Matrix-Vector multiplication

```
#pragma omp parallel for default (none) shared (m,n,a,b,c) private (i,j,sum)
```



The default clause

- default (none | shared | private)
  - none
    - No implicit defaults.
    - Have to scope all variables explicitly.
  - shared
    - All variables are shared.
    - The default in absence of an explicit “default” clause.
  - private
    - All variables are shared.

The reduction clause

- reduction (operator : list)
  - Reduction variable(s) must be shared variables.
  - Operators:
    - +, -, *, max, min, …

```
int maximum = 0, i;
int Vector[1000];
#pragma omp parallel for shared (Vector) private (i) reduction (max : maximum)
{
        for (i=0; i<1000; i++)
            if (Vector[i] > maximum)
                maximum = Vector[i];
}
```

The ordered clause

- The iterations of the loop are executed in the same order than the sequential execution (1 thread at a time).

```
#pragma omp parallel for ordered
{
    for (i=0; i<n; i++)
        printf (“n = %d\d, n);
}
```

Load balancing

- Load balancing is an important aspect of performance.
- For regular operations (e.g. a vector addition), load balancing is not an issue.
- For les regular workloads, care needs to be takes in distributing the work over the threads.
- Irregular workloads include transposing a matrix, multiplication of irregular matrices, …
- The schedule clause supports various iteration scheduling algorithms.

The schedule clause

- schedule (static | dynamic | guided [, chunk]) (runtime)
  - static [, chunk]
    - Distribute iterations in blocks of size “chunk” over the threads in a round-robin fashion.
    - In absence of “chunk”, each thread executes approx. N/P iterations.
  - dynamic [, chunk]
    - Fixed portions of work; sixe is controlled by the value of chunk.
    - The default in absence of an explicit “default” clause.

The schedule clause

- schedule (static | dynamic | guided [, chunk]) (runtime)
  - guided [, chunk]
    - Same dynamic behavior as “dynamic”, but size of the portion of work decreases exponentially.
  - runtime
    - Iteration scheduling is set at runtime through environment variable OMP_SCHEDULE.

The schedule clause

- schedule (static | dynamic | guided [, chunk]) (runtime)
  - guided [, chunk]
    - Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially.
  - runtime
    - Iteration scheduling is set at runtime through environment variable OMP_SCHEDULE.

The schedule clause example

- 80 iterations on 4 threads

TID	0	1	2	3
Static No chunk	0-19	20-39	40-59	60-79
Static Chunk 10	0-9 40-49	10-19 50-59	20-29 60-69	30-39 70-79
Dynamic Chunk 10	0-9	10-19 40-49 60-69	20-29 70-79	30-39 50-59
Guided Chunk 10	0-19	20-34 57-66	35-46 67-76	47-56 77-79

The nowait clause

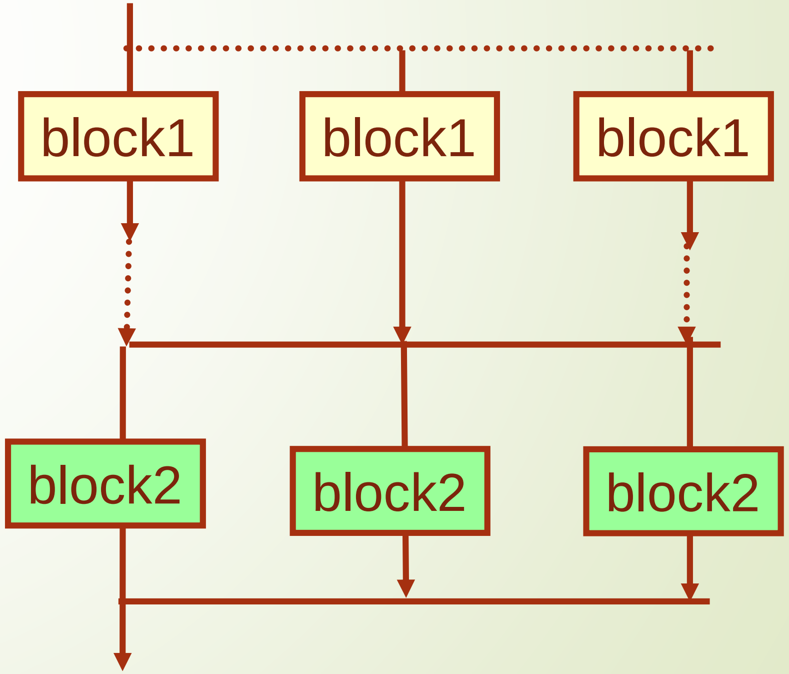
- To minimize synchronization, some OpenMP pragmas support the nowait clause.
- If present, threads will not synchronize/wait at the end of that particular construct.

```
#pragma omp parallel for default (none) shared (n,a,b) private (i) nowait
{
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;
}
```

The omp barrier directive

- All threads wait until the last arrives to the barrier

```
#pragma omp parallel num_threads (3)
{
    Block1;
    #pragma omp barrier
    Block2;
}
```



The omp sections directive

- Must be in a parallel region.
- Sections are distributed among threads.
- Each thread executes a different section.
- Allows task level parallelism.
- SECTION pragma defines each section.

The omp sections directive - example

```
#pragma omp parallel num_threads (3)
{
    #pragma omp sections
    {
        #pragma omp section {
                Block1;
        }
        #pragma omp section {
                Block2;
        }
        #pragma omp section {
                Block3;
        }
    }
}
```

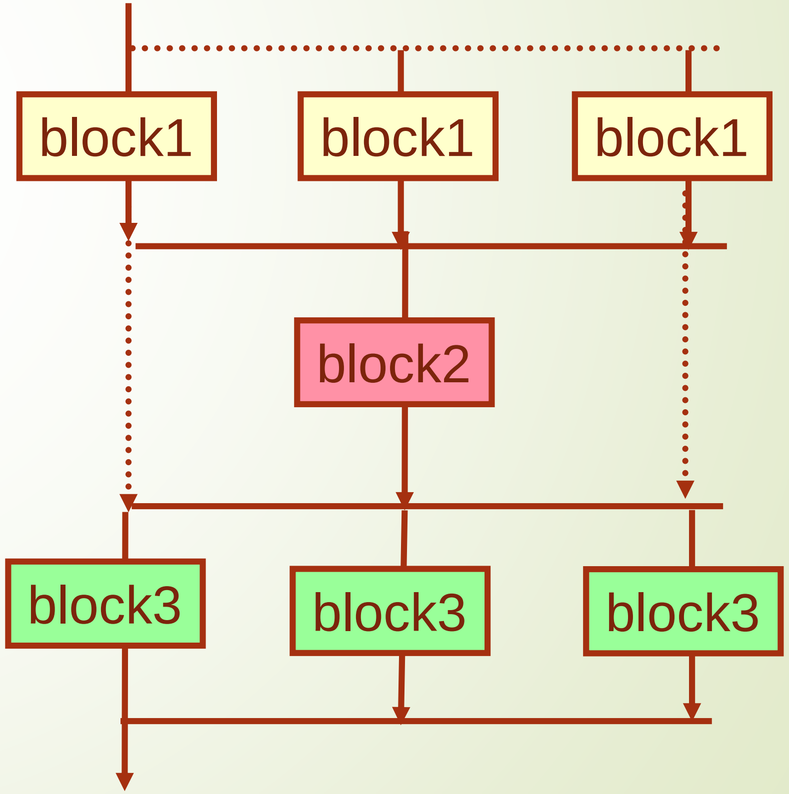
The omp sections directive - example

```
#pragma omp parallel default (none) shared (n,a,b,c,d) private (i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
            for (i=0; i<n-1; i++)
                b[i] = (a[i] + a[i+1])/2;
        #pragma omp section
            for (i=0; i<n-1; i++)
                d[i] = 1.0/c[i];
    }
}
```

The omp single directive

- The code included in the single section will be executed only by one thread.

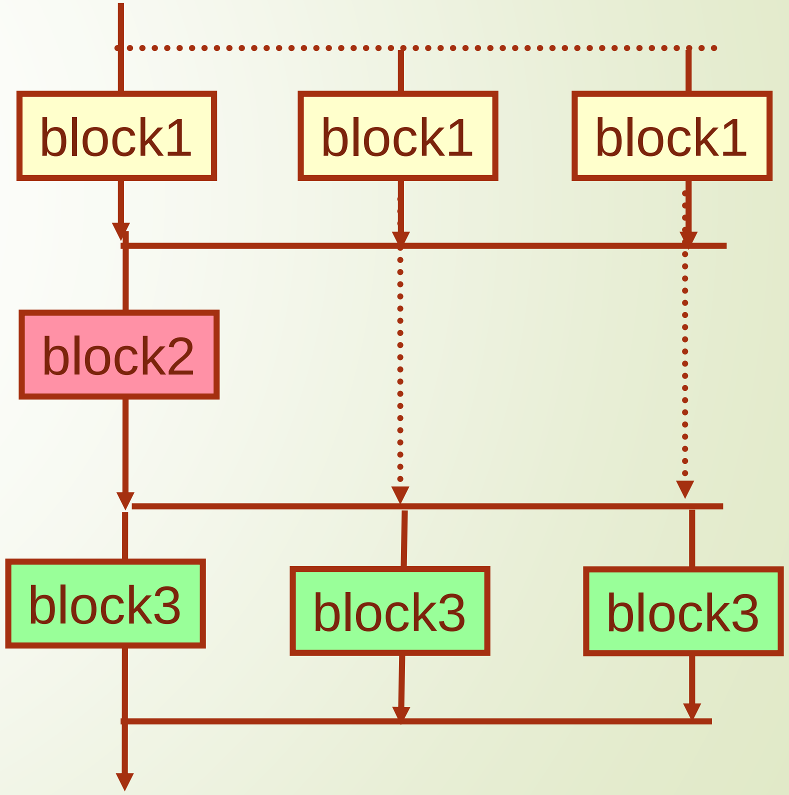
```
#pragma omp parallel num_threads (3)
{
    Block1;
    #pragma omp single {
            Block2;
    }
    Block3;
}
```



The omp master directive

- The code included in the master section will be executed only by master thread.

```
#pragma omp parallel num_threads (3)
{
    Block1;
    #pragma omp master {
            Block2;
    }
    Block3;
}
```



The omp critical directive

- A critical section is a block of code which can be executed by only one thread at a time.
- Can be used to protect updates to shared variables.

```
sum = 0.0;
#pragma omp parallel for shared(sum, A, B) private (i)
{
    for (i=0; i<N; i++{
        A[i] = B[i]/2;
        #pragma omp critical
            sum += A[i];
    }
}
```

OpenMP 3.0

- TASK Construct
  - The TASK construct defines an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread in the team.
  - The data environment of the task is determined by the data sharing attribute clauses.

```
#pragma omp task [clause ...]
```

- The TASKWAIT construct specifies a wait on the completion of child tasks generated since the beginning of the current task.

```
#pragma omp taskwait
```

The task directive - example

```
struct node {node *next, .../*data*/}
void process_list_items (node* head)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (node* p = head; p; p = p->next)
            {
                #pragma omp task firstprivate(p)
                    process(p);
            }
        }
    }
}
```

OpenMP 3.0

- TASKGROUP Construct
  - The TASKGROUP construct specifies a wait on completion of child tasks of the current task and their descendent tasks
  - A TASKGROUP region binds to the current task region. The binding thread set of the taskgroup region is the current team
- TASKYIELD Construct
  - The TASKYIELD construct specifies that the current task can be suspended in favor of execution of a different task

OpenMP 4.0

- Support thread affinity policies (proc_bind, get_proc_bin, OMP_PLACES)
- Support execution on devices (accelerators) (omp_set_default_device, omp_get_default_device, omp_get_num_devices, omp_get_num_teams, omp_get_team_num, and omp_is_initial_device)
- Reduction clause extended to support user defined reductions
- The concept of cancellation is added

Run-time Library

- OMP_SET_NUM_THREADS (SCALAR)
  - Sets the number of threads that will be used in the next parallel region.
  - Only works if called from a sequential portion of the program.
- OMP_GET_NUM_THREADS ()
  - Returns the number of threads in the parallel region where it's called.
  - The default number of threads depends on the application.
- OMP_GET_THREAD_NUM ()
  - Returns the thread id of the thread that calls it.
  - Master thread has id 0.

Run-time Library

- omp_in_parallel
- omp_set_dynamic
- omp_get_dynamic
- omp_set_nested
- omp_get_nested
- omp_set_schedule
- omp_get_schedule
- omp_get_thread_limit
- omp_set_max_active_levels
- omp_get_max_active_levels
- …

References

- Ruud van der Pas “An Introduction Into OpenMP”. Sun Microsystems. iWOMP 2005.
- OpenMP Architecture Review Board. OpenMP Application Program Interface. Version 2.5. May 2005.
- OpenMP Application Program Interface. Version 4.0. July 2013
- www.openmp.org
- computing.llnl.gov/tutorials/openMP/